MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 750 November, 1983

Design Issues in Parallel Architectures for Artificial Intelligence

Carl Hewitt
Henry Lieberman

Abstract

Development of highly intelligent computers requires a conceptual foundation that will overcome the limitations of the von Neumann architecture. Architectures for such a foundation should meet the following design goals:

* Address the fundamental organizational issues of large-scale parallelism and sharing in a fully integrated way. This means attention to organizational principles, as well as hardware and software.
* Serve as an experimental apparatus for testing large-scale artificial intelligence systems.
* Explore the feasibility of an architecture based on abstractions, which serve as natural computational primitives for parallel processing. Such abstractions should be logically independent of their software and hardware host implementations.

In this paper we lay out some of the fundamental design issues in parallel architectures for Artificial Intelligence, delineate limitations of previous parallel architectures, and outline a new approach that we are pursuing.

This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Major support for the research reported in this paper was provided by the System Development Foundation. Major support for other related work in the Artificial Intelligence Laboratory is provided, in part, by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505.

1. Introduction

Large-scale distributed systems are evolving rapidly because of the following developments:

* The burgeoning growth of personal computers;

* The development of local and national electronic networks;

* The widespread requirement of *arm's-length* (discussed below) transactions among agencies.

We envision the development of open systems [Hewitt and de Jong 83a, Hewitt and de Jong 83b]with the following properties:

* *Continuous change and evolution.* Distributed artificial intelligence systems are always adding new computers, users, and software. As a result, systems must be able to change as the components and demands placed upon them change. Moreover, they must be able to evolve new internal components in order to accommodate the shifting work they perform. Without this capability, every system must reach the point where it can no longer be expanded to accommodate new users and uses.

* *Absence of bottlenecks and decentralized decision-making.* A bottleneck is a channel through which all communications must flow. In the von Neumann architecture, the single path between processor and memory acts as such a bottleneck. An adequate architecture for large-scale artificial intelligence systems cannot have such bottlenecks. A centralized decision-making mechanism would inevitably become a bottleneck; therefore decision-making must be decentralized.

* *Arms-length relationships and decentralized decision-making.* In general, the computers, people, and agencies that make up open systems do not have direct access to one another's internal information. For example, suppliers and their customers or competitors must deal with one another at *arm's length.* Arms-length relationships imply that the architecture must accommodate multiple computers at different physical sites that do not have access to each others internal components. They also imply that the decision making is decentralized.

* *Perpetual inconsistency among knowledge bases.* Because of privacy and discretionary concerns, different knowledge bases will contain different perspectives and conflicting beliefs. Thus all the knowledge bases of a distributed artificial intelligence system taken together will be perpetually inconsistent. Decentralization also makes it impossible to update all knowledge bases simultaneously. This implies that it is not even possible to know what kinds of information are contained in all the local knowledge bases in the system at any one time. Systems must thus be able to operate in the presence of inconsistent and incomplete knowledge bases.

* *Need for negotiation among system components.* In a highly distributed system, no

system component directly controls the resources of another. The various components of the system must persuade one another to provide capabilities. A distributed artificial intelligence system's architecture therefore must support a mechanism for negotiation among components.

Continuous growth and evolution, absence of bottlenecks, arm's-length relationships, inconsistency among knowledge bases, decentralized decision making, and the need for negotiation are interdependent and necessary properties of open systems.

2. Foundational Issues

Open systems raise important foundational issues in Artificial Intelligence including the following:

* *Integration of action, description, and reasoning.* Three kinds of computer languages have been developed: the descriptive, the action-oriented, and the reasoning or deductive. Descriptive languages represent knowledge using descriptions: languages that are based broadly on this approach include first order logic, KRL, NETL, Omega, KL-ONE, etc. Action-oriented programming languages such as Fortran, Lisp and Ada implement processes that change objects. They tend to provide relatively weak descriptions of objects and the relationships among them. Reasoning languages plan, solve problems, or prove theorems. Logic programming languages (such as Prolog [Kowalski 83]) are a subset of the reasoning languages that are limited by cumbersome description and reasoning capabilities and by having inelegant mechanisms for taking action [Hewitt 83]. Historically, languages strong in one of these kinds of capacities have been relatively weak in the others. All three capabilities: action, description, and reasoning are necessary for large-scale artificial intelligence systems.

* *Inadequacy of the closed-world assumption.* The *closed world assumption* is that the information about the world being modeled is complete in the sense that exactly those relationships that hold among objects are derivable from the local information possessed by the system. Systems that make use of the closed world assumption (such as Planner [Hewitt 69] and Prolog [Kowalski 74]) typically assume that they can find all existing instances of a concept by searching their local storage. At first glance it might seem that the closed world assumption, almost universal in the artificial intelligence and database literature, is smart because it provides a ready default answer for any query. Unfortunately the default answers provided become less realistic as open systems increase in size since less of the information is available locally.

* *Empirical Knowledge.* Algorithmic procedures are those about which properties such as efficiency can be proved mathematically, e.g. Gaussian Elimination, Quick Sort, etc. In contrast Artificial Intelligence is mainly concerned with empirical procedures. In general empirical procedures make essential use of empirical knowledge and require interaction with the physical world in real time. For example a person's procedures for

driving from Palo Alto to Boston rely on a great deal of empirical knowledge, e.g. the relationship between a road map and a view from the car of the terrain on the map. Interaction with the physical world in real time is required at many points, e.g. when a traffic light turns yellow. Few interesting properties can be mathematically proved about a person's procedures for driving across the country. Thus driving across the country provides prime examples of empirical procedures.

* *Dynamic Allocation.* An artificial intelligence architecture needs to dynamically allocate and re-allocate computational resources as required by ongoing empirical computations. Empirical computations require dynamic allocation because it is impossible to plan in advance which procedures will need to be run. Artificial intelligence applications require that the highest priority be short latency (fast response time) in order to efficiently dynamically re-allocate resources as needed.

3. Limitations of Previous Architectures for Artificial Intelligence

Von Neumann machines, being sequential, are *inadequate* vehicles on which to base the development of large-scale artificial intelligence systems. We need architectures that are inherently parallel and sufficiently general to meet the requirements of open systems. Existing languages, which were designed for existing architectures, tend to be similarly inadequate. Languages designed for von Neumann machines are inherently sequential. Extensions to these languages unnecessarily restrict the amount of concurrency.

Artificial intelligence systems need to organize information using *description lattices* that are the further development of early ideas on "semantic memories" [Quillian 68]. The consistency and completeness properties of first-order description lattice logics have been systematically explored [Attardi and Simi 81]. One of the most important design criteria for a parallel architecture for Artificial Intelligence is to be able to store, retrieve, and reason about information in description lattices in parallel.

Almost all previous non von Neumann architecture projects are based on the unexamined assumption of *linear scaling*: their announced goal is to build an architecture in which *the speed of computation will increase in direct proportion to the number of computing elements without any reprogramming.* Almost all non von Neumann architectures have uncritically accepted the assumption that simple linear scaling is possible as a criterion for success.

Of course simple linear scaling is theoretically impossible because communication delay increases as a machine becomes physically larger. As Chuck Seitz has pointed out [Seitz 83], the cases in

which linear scaling is *almost* achieved involve rather special cases of crystalline or systolic regularity. For open systems, we feel that linear scaling is not directly achievable because increasing the number of computational elements in a system increases the complexity of the relationships among them.

Use of the principles by which work is organized in large-scale human organizations is a powerful tool for making significant progress in architectures for Artificial Intelligence. The issues involved in the organization of work must be dealt with in the early stages of architectural design because it can have a significant impact on the kind of architecture that is useful. For example the architecture must support capabilities like having sponsors for allocating resources [Kornfeld 82, Barber 82].

To deal with this issue, we are working on the development of a *computational* theory of organizations and the organization of work [Barber, de Jong, and Hewitt 83]. *An architecture that is capable of dynamic growth must be able to undertake major reorganizations of the work of computations as they grow in size.* In order to do this, the system must have an understanding of its own capabilities and limitations as well as an understanding of the goals and constraints of the computations being performed.

Below we consider particular limitations of previously proposed architectures in the context of large-scale parallel artificial intelligence applications. Most of the limitations stem from the fact that previous architectures were not designed for highly parallel artificial intelligence applications in open systems.

3.1. Applicative, Functional, and Reduction Architectures.

Applicative, functional, and reduction machine organizations are based on the mathematical theory of functions and function spaces. The absence of assignment commands and state change facilitates parallelism. Applicative, functional, and reduction architectures are based underlying mathematical theory of the lambda calculus [Church 41] which provides a very elegant proof theory [Scott 72] and mathematical model of computation.

An important limitation of the lambda calculus class of languages is that they do not support large-scale sharing of objects with a changing local state. Shared objects with a changing local state are required to support the allocation of resources and the implementation of cooperating and

competing subsystems for artificial intelligence applications [Hewitt and de Jong 83b].

3.2. Data Flow Architectures.

Historically the term "data flow" is derived from trying to better structure the flow of data and communication signals between hardware modules. Modeling software in terms of data and communication between hardware modules has turned out to be limited because of the dynamic creation of objects that occurs in software. To cope with the limitation, Data Flow machine organizations are gradually evolving to have much in common with applicative, functional, and reduction machines.

Data Flow architectures exploit the nature of algorithmic procedures to plan and map algorithms efficiently onto the hardware before computation begins. As a result Data Flow architectures place highest priority on thruput as a criterion for efficient execution of the planned procedures. They were developed for running algorithmic rather than empirical procedures. Focusing on algorithmic procedures, they place great emphasis on being able to plan the resource allocation for a computation before starting. They are weak in the capabilities for dynamic allocation of computational resources such as parallel garbage collection [Lieberman and Hewitt 83]. Architectures for Artificial Intelligence require a more dynamic allocation of resources and greater emphasis on short latency (fast response time).

3.3. Logic Programming Architectures.

Logic Programming is based on mathematical logic and the quantificational calculus. The quantificational calculus is a well-defined powerful description language with solid elegant mathematical foundations. One of the major findings of our research is that Logic Programming architectures will not be adequate for distributed, parallel, artificial intelligence systems. Logic Programming is inadequate for dealing reliably with empirical knowledge [Hewitt and de Jong 83a, Hewitt 83]. Current Logic Programming languages such as Prolog [Kowalski 83] are based on essentially the same technology as Planner-like languages [Hewitt 69, Sussman, Winograd, and Charniak 70]. Prolog [Kowalski 83] makes use of a flat data base of assertions that does not provide an adequate organization of the knowledge in the form of description lattices. Logic Programming has further problems in that it confounds issues of description and action [Hewitt and de Jong 83b, Hewitt 83].

3.4. Blackboard Architectures.

The name "blackboard" brings to mind a group of agents communicating with a blackboard. This does not work work well in practice because the blackboard quickly becomes a bottleneck in reading and writing information for large scale parallel systems. Introducing multiple blackboards quickly shifts the emphasis to some other architectural paradigm such as message passing.

3.5. SIMD Architectures.

Arrays of single instruction, multiple data (SIMD) stream computers are important for the early stages of visual, speech, and tactile processing. Thus an Artificial Intelligence architecture needs to be able to interface with such arrays on the periphery. SIMD architectures are designed to perform one operation at a time on a large array of data. Therefore they are not suitable of doing large-scale symbolic reasoning which requires that multiple symbolic operations be performed in parallel on different symbolic structures.

3.6. Global Shared Memory Architectures.

A global shared memory is a single large shared structure which is used for interprocess communication. Global Shared Memory machine organizations communicate at a very low level reading and writing individual words of memory. Such architectures are inadequate for physically distant computers, because the central memory acts as a bottleneck. A global shared memory will become increasingly less attractive because the ratio of the average latency for accessing the global shared memory relative to the clock period will gradually increase over the course of time as communication time becomes a dominant efficiency consideration. Because of the need for dynamic allocation of resources, parallel artificial intelligence systems require a low relative latency (fast response time) for effective operation.

4. Actors

Actor theory [Hewitt and Baker 77, Clinger 81] provides a foundation for addressing the problems of constructing distributed, large-scale artificial intelligence systems. Actor theory treats issues of scaling information processing systems in an integrated fashion: It applies equally to large-scale multi-processing machines that reside in a single room, and to those in which the machines are geographically dispersed. Actors are inherently parallel. An actor is defined by its behavior when it processes communications. When an actor processes a communication it can perform the following four kinds of primitive actions concurrently:

1. Make simple decisions
2. Create more actors
3. Send more communications
4. Specify how it will behave in response to the next message received, thus characterizing its new local state

Conceptually, there is no *a priori* limit on the computational power of an actor, e.g. a single actor can implement a whole microprocessor system. In practice, we make each actor relatively simple in order to keep systems modular. The power of actor systems stems from the ease with which systems of more specialized actors can be combined to accomplish large tasks.

In a theoretical sense, actor systems are more powerful than lambda calculus [Church 41] systems. Consider the problem of when given a formula in formal system (which may or may not be derivable), either deriving the formula or giving up and reporting failure. For any given lambda calculus program, there exists a bound on how deep the search will proceed. However, we can easily construct an actor system which will always prove the formula or report failure but for which there is no bound as to how deeply it will search. The actor system can be constructed with two actors which we will call the Sponsor and the Searcher. The Searcher commences a search for a derivation of the formula which is potentially nonterminating. The Sponsor counts up to some fixed number and then sends a stop message to the Searcher. However since the stop message might be arbitrarily delayed, there is no bound on how deep the search might proceed. The above example illustrates a *qualitative* difference between the lambda calculus class of systems and actor systems. It is similar to the kind of qualitative difference that exists between finite and infinite state machines. Such qualitative differences have important effects on the style of development of artificial intelligent applications.

The utility and versatility of actors stem from the following properties:

* *Actor systems are inherently parallel.* Because each actor responds to messages independently of other actors, systems of actors inherently have a high degree of parallelism. Different actors can be processing sub-tasks simultaneously, coordinating their activities by passing messages among themselves.

* *Actors have hardware generality.* Since actors are virtual computational units, their implementation is not dependent upon particular hardware configurations such as machine boundaries, the number of processors, or the physical location of machines.

An actor can be implemented in hardware, in micro-code, or in software. Actors are organized by load-balancing to spread processing load and communication delays, by migration to relieve overcrowding, and by efficient, real-time, distributed garbage collection to improve locality of communication and to recover the use of storage occupied by inaccessible actors.

* *Actors have software generality.* Being defined mathematically [Clinger 81], actors are independent of any programming language. Actors are very general with respect to the languages and language features they can support. In particular, they provide a uniform basis for description, action, and reasoning [Barber, de Jong, and Hewitt 83]. Actors support the parallel processing of lattice networks of descriptions (e.g. semantic networks), pattern-directed rules for reasoning, as well as procedures for taking action.

5. The Apiary Approach

We are developing an experimental machine architecture, called the Apiary, based on actor theory [Hewitt 80]. The goal is to create an integrated, parallel hardware-software system that has the generality required for large-scale artificial intelligence systems. To date, much of the implementation work on the Apiary has centered around *simulating* the Apiary on a network of current-generation sequential computers [Lieberman 83].

Most simulators for new machines perform at the level of *instruction sets* or *virtual machines.* The simulator implements a sequential interpreter for the instruction set in the host language, and programs written using the new machine's instruction set can be tested. Compilers translate higher-level languages into the instruction set. Instruction sets for conventional machines usually specify operations like loading and storing registers with fixed length bit strings. The instruction set level has the advantage that it is high enough to be convenient as the target for problem-solving languages, yet low enough to experiment with the algorithms which will be needed to realize the computational model in hardware.

Instead, we model the Apiary as a set of *workers*, each worker being analogous to a single computer executing instructions serially, together with its own memory, and the ability to communicate with other workers. The Apiary simulator must bridge the gap between programming sequential processors with modifiable state and a multi-processor system with no global state.

The primitive operations of the Apiary are not like those of conventional machines. Whereas instructions in Von Neumann machines manipulate the contents of registers, the Apiary must deal

solely with actors and their responses to messages. Although an Apiary may be implemented at the lowest hardware level with transistors and wires, this must be hidden from Apiary applications. The four primitive capabilities mentioned earlier (creating actors, sending messages, making decisions, and changing behavior) form the backbone of a virtual machine for an Apiary worker, together with operations on primitive data types such as addition of numbers.

We have not yet taken the Apiary simulator down to the level of coding instructions as bit strings, defining memory formats, and specifying register sets. Though this will eventually be necessary for hardware implementation, the details will depend heavily on the particulars of implementation technology.

The Apiary is a parallel problem solving system, where users write programs whose computations may range over many processors. Artificial intelligence applications are characterized by their unpredictable nature. One cannot predict in advance how many concurrent activities will be necessary or desirable for solving a problem, how much memory will be needed, or how to best divide work among available processors. It follows that a parallel system for Artificial Intelligence must dynamically allocate all processor, communication, and memory resources in the system, without explicit intervention by applications programs.

The Apiary architecture accommodates two broad classes of machines: core machines and periphery machines. The core consists of high performance processors connected with high bandwidth, low latency communication links. The periphery consists of less powerful processors that are portable and do not require air conditioning. The hardware will be composed of a changing number of physical processors, executing specific programs, each with its own local memory, yet we do not want programs to depend on the specifics of the hardware configuration. Our goal is to create an illusion of flexible parallel computation given inflexible serial hardware.

Most of the work involved is the responsibility of a set of *housekeeping algorithms*, and much of the challenge of the implementation of the Apiary will lie in the creation of simple and efficient algorithms for tasks of real-time such as the following:

* *Real-time efficient garbage collection.* Like traditional systems that support dynamic allocation of storage, the Apiary will require real-time garbage collection that is efficient. Our preliminary thoughts on this important issue are spelled out in more detail in [Lieberman and Hewitt, CACM 1983]. Efficiency of storage recovery is measured by storage latency (the average time which elapses between when an actor

becomes inaccessible and when its storage is recovered) and the processing required (the average processing cost in recovering the storage of an inaccessible actor).

* *Balancing.* The Apiary supports activities and actors at multiple sites. Without corrective action it could easily get out of balance with some of the sites having insufficient storage, communications, and/or processing power. We are designing mechanisms that will promote balance and tend to restore the Apiary to balance.

* *Locality.* Achieving locality of communication is fundamental to the efficiency of the Apiary. We conjecture that communication bandwidth between two points in the Apiary needs to fall off exponentially with their distance.

6. Apiary Software Foundations

Developing a new architecture for large-scale artificial intelligence systems is largely a software problem. Prelude is the name of the software system we are developing as the foundation for the Apiary. The goal is for Prelude to support unified description, action, and reasoning systems that can exploit the large-scale parallelism made possible by VLSI.

We have developed a number of experimental software systems dealing with different aspects of the design of high-level actor-based language systems. First a basic actor programming language [Lieberman 81] was developed for sending communications, creating actors, and making local state changes. Next a description language that incorporates the descriptive capabilities of logic in the in the context of lattices of descriptions [Hewitt, Attardi, and Simi 80] and a reasoning system that provides for reasoning about beliefs and goals in parallel was developed [Kornfeld and Hewitt 81, Kornfeld 82]. The description system was characterized axiomatically [Attardi and Simi 81]. Then the reasoning system was re-implemented to incorporate the use of the description system [Barber 82]. These languages and systems were independently implemented in somewhat incompatible ways. We plan to integrate their capabilities using Act2 [Theriault 83]. Currently Act2 integrates communication and local change capabilities with the lowest level descriptions in the description system. Being inherently parallel and having no assignment commands, it has proven to be well suited for the implementation of asynchronous concurrent systems.

The generality of Act2 has been demonstrated by a metacircular description of Act2 in Act2 [Theriault 83]. Part of the demonstration consisted of constructing parallel applicative interpreters directly, such as one for Pure Lisp with parallel argument evaluation. This cannot be done using the lambda calculus [Church 41] class of interpreters themselves, for they lack the ability to express

the required local state changes. Thus, we discovered, through a concrete example, that Act2 is more powerful than the lambda calculus class of languages. More generally, Act2 provides good support for control-driven, data-driven, and demand-driven styles of computation.

In a related development, investigation of a shared financial account example confirmed the expectation that Act2 is more suitable than the lambda calculus class of languages for dealing with computational problems involving shared actors with changing local states. While working with the shared financial account example, we implemented a new approach to actor state changes [Hewitt, Attardi, and Lieberman 79, Hewitt and de Jong 83b]. From the perspective of this new approach, actors change state by transforming themselves into other actors.

Work is proceeding on the development of a source language debugging system for parallel systems called Time Traveler. The Time Traveler system builds on previous work on debugging hardware [Giaimo 75], debugging sequential programs [Balzer 69], and parallel simulation [Jefferson and Sowizral 82].

7. Conclusion

In this paper we have explored some of the fundamental design issues in parallel architectures for Artificial Intelligence, explained limitations of previous parallel architectures, and outlined the Apiary architecture that we are pursuing. The Apiary architecture has proven to be an excellent vehicle for exploring the ideas and theories presented in this paper. Demonstration of the practical importance of the Apiary awaits the allocation of suitable resources for realistic testing and for the development of applications.

8. Acknowledgments

Much of the work underlying our ideas was conducted by members of the Message Passing Semantics group at MIT. We especially would like to thank Jon Amsterdam, Jerry Barber, Peter de Jong, Tim McNerney, Elijah Millgram, Chunka Mui, Tom Reinhardt, and Dan Theriault. Valuable feedback was provided by Toni Cohen, Mike Farmwald, Morven Gentleman, Fanya Montalvo, Chuck Seitz, and Henry Sowizral. Comments by Gul Agha, Jerry Barber, Mike Brady, James Cerrato, Toni Cohen, Peter de Jong, Elihu Gerson, Maria Gruenwald, Steve Kossar, Fanya Montalvo, Tom Reinhardt, Chuck Seitz, Charles Smith, and Leigh Star have been of fundamental importance in improving the organization and content of this paper.

This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Major support for the research reported in this paper was provided by the System Development Foundation. Major support for other related work in the Artificial Intelligence Laboratory is provided, in part, by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505. We would like to thank Charles Smith, Mike Brady, Patrick Winston, and Gerald Wilson for their support and encouragement.

References

[Attardi and Simi 81] Attardi, G. and Simi, M. Semantics of Inheritance and Attributions in the Description System Omega. Proceedings of IJCAI 81, IJCAI, Vancouver, B. C., Canada, August, 1981.

[Balzer 69] Balzer, R. M. EXDAMS--EXtendable Debugging and Monitors Systems. Spring Joint Computer Conference, AFIP, 1969.

[Barber 82] Barber, G. R. Office Semantics. Ph.D. Th., Massachusetts Institute of Technology, 1982.

[Barber, de Jong, and Hewitt 83] Barber, G. R., de Jong, S. P., and Hewitt, C. Semantic Support for Work in Organizations. Proceedings of IFIP-83, IFIP, Sept., 1983.

[Church 41] Church, A. The Calculi of Lambda-Conversion. In *Annals of Mathematics Studies Number 6*, Princeton University Press, 1941.

[Clinger 81] Clinger, W. D. Foundations of Actor Semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May, 1981.

[Giaimo 75] Giaimo, E. C. III. The Program Monitor: A Programmable Instrument for Computer Hardware and Software Performance Measurement. Master Th., Massachusetts Institute of Technology, 1975.

[Hewitt 69] Hewitt C. E. PLANNER: A Language for Proving Theorems in Robots. Proceedings of IJCAI-69, IJCAI, Washington D. C., May, 1969.

[Hewitt 80] Hewitt C. E. The Apiary Network Architecture for Knowledgeable Systems. Conference Record of the 1980 Lisp Conference. Stanford University. Stanford. California. August, 1980.

[Hewitt 83] Hewitt, C. Some Fundamental Limitations of Logic Programming. A.I. Memo 748, MIT Artificial Intelligence Laboratory, November, 1983.

[Hewitt and Baker 77] Hewitt, C. and Baker, H. Laws for Communicating Parallel Processes. 1977 IFIP Congress Proceedings, IFIP, 1977.

[Hewitt and de Jong 83a] Hewitt, C., de Jong, P. Open Systems. Perspectives on Conceptual Modeling, Springer-Verlag, 1983.

[Hewitt and de Jong 83b] Hewitt, C., de Jong, P. Analyzing the Roles of Descriptions and Actions in Open Systems. Proceedings of the National Conference on Artificial Intelligence, AAAI, August, 1983.

[Hewitt, Attardi, and Lieberman 79] Hewitt C., Attardi G., and Lieberman H. Specifying and Proving Properties of Guardians for Distributed Systems. Proceedings of the Conference on Semantics of Concurrent Computation, INRIA, Evian, France, July, 1979.

[Hewitt, Attardi, and Simi 80] Hewitt, C., Attardi, G., and Simi, M. Knowledge Embedding with a Description System. Proceedings of the First National Annual Conference on Artificial Intelligence, American Association for Artificial Intelligence, August, 1980.

[Jefferson and Sowizral 82] Jefferson, D., Sowizral, H. Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control. Tech. Rep. N-1906-AF, RAND, December, 1982.

[Kornfeld 82] Kornfeld, W. Concepts in Parallel Problem Solving. Ph.D. Th., Massachusetts

Institute of Technology, 1982.

[Kornfeld and Hewitt 81] Kornfeld, W. A. and Hewitt, C. The Scientific Community Metaphor. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11*, 1 (January 1981).

[Kowalski 74] Kowalski, R. A. Predicate Logic as Programming Language. Proceedings of IFIP-74, IFIP, 1974.

[Kowalski 83] Kowalski, R. A. Logic Programming. Proceedings of IFIP-83, IFIP, 1983.

[Lieberman 81] Lieberman, H. A Preview of Act-1. A.I. Memo 625, MIT Artificial Intelligence Laboratory, 1981.

[Lieberman 83] Lieberman, H. An Object-Oriented Simulator for the Apiary. Proceedings of AAAI-83, AAAI, Washington, D. C., August, 1983.

[Lieberman and Hewitt 83] Lieberman, H. and Hewitt, C. A Real Time Garbage Collector Based on the Lifetimes of Objects. *CACM 26*, 6 (June 1983).

[Quillian 68] Quillian, M. R. Semantic Memory. In *Semantic Information Processing*, Minsky, M., Ed., MIT Press, 1968.

[Scott 72] Scott, D. S. Lattice Theoretic Models for Various Type-free Calculi. Proceedings 4th International Congress in Logic, Methodology and the Philosophy of Science, Bucharest, Hungary, 1972.

[Seitz 83] Seitz, C. Private communication.

[Sussman, Winograd, and Charniak 70] Sussman, G. J., Winograd, T., and Charniak, E. MICRO-PLANNER Reference Manual. AI Memo 203, MIT AI Lab, 1970.

[Theriault 83] Theriault, D. Issues in the Design and Implementation of Act2. Technical Report 728, MIT Artificial Intelligence Laboratory, June, 1983.